Artifact evaluation instructions

Getting started guide

Intro

This artifact is prepared as a docker image. The docker image contains the `Eff` compiler built from source along with additional external benchmark files and working version of multicore OCaml compiler used for running benchmarks. The dockerfile was tested on MacOS (version v19.03.13-beta2 and v20.10.7), Ubuntu (20.04.2 LTS), and Windows (20.10.6, build 370c289). Make sure that the docker settings allow for at least 6GB of ram.

Instructions

- Install docker for your operating system, any recent enough version will do. This artifact has been tested with Docker version 20.10.2, build 20.10.2-0ubuntu1~20.04.2.
- Before any operation, make sure that `docker-deamon` or docker service is running.
- Obtain Docker image. This can be done in two ways:

  i. Use prebuilt image available on Dockerhub (faster, recommended):

  - Pull image using `docker pull j00sko/eff-oopsla-artifact:oopsla2021-update` .

  ii. Build image locally(slower).

  - The repository contains the same `Dockerfile` used to build the image on dockerhub.
  - Run `docker build -t j00sko/eff-oopsla-artifact:oopsla2021-update .` . Build can take up to 30 minutes depending on your internet connection and computer power. Warnings from opam that "running as root is not recommended" are expected and should be ignored.

- Test successful build by running `docker run j00sko/eff-oopsla-artifact:oopsla2021-update ./eff.exe examples/choice.eff` . This runs a simple example and should produce:

```
[WARNING] Running as root is not recommended
- : int = 10
- : int = 10
val choose_all : 'a => 'a list = <handler>
- : int list = [10; 5; 20; 15]
val choose_all2 : 'a => 'a list = <handler>
- : int list list = [[10; 5]; [20; 15]]
- : int list list = [[10; 20]; [5; 15]]
- : int list list = [[10; 20]; [10; 15]; [5; 20]; [5; 15]]
```

The example runs a simple `eff` program and outputs the results.

Warnings about running as root can safely be ignored.

Step by Step Instructions

Intro

Dockerfile starts with an original OCaml docker image and installs necessary libraries for building the `Eff` compiler presented in the paper. Pinned version of `Eff` is compiled from scratch and an executable is created. Finally an additional switch for Multicore OCaml on which benchmarks are run is installed and set as default.

Pinned version of `Eff` is practically the same as the one used for the paper with some minor differences:

- Library `delimcc` is excluded since it doesn't work with Multicore OCaml.
- Benchmarks from `Handlers in action` and `Eff in Ocaml` are commented out. These benchmarks are a few orders of magnitude slower than the ones presented in the paper and provide no additional information, while greatly increasing benchmarking time.
- Additional flag is added to optionally compile Multicore OCaml benchmarks on both versions `4.10` (already in paper) and `4.12` (newly added).
- Bash script helpers are added to enable easier access to functionality and artifact testing.
- Number of runs for benchmarks is lowered due to overhead induced by container. Number of runs can be increased by setting parameters in `misc/code-generation-benchmarks/display-results/benchmark.ml` in line 12 for human readable output and `misc/code-generation-benchmarks/generate-graphs/graphs.ml` in lines 6 and 8 for graph generation.

Any commands can also be executed directly in source code repo available at https://github.com/matijapretnar/eff/tree/explicit-effect-subtyping, provided that you have OCaml with necessary libraries installed (following instructions in eff repo or Dockerfile).

Execute command `docker run -it j00sko/eff-oopsla-artifact:oopsla2021-update bash`, to get access to docker.

Dockerfile is minimal and `nano` is the only available editor. You can install an editor of your choice `apt-get install vim`, but be aware, that any changes made to file system will be undone once you exit the container (including installing a new editor).

Available commands

Unless otherwise stated, all commands are run in docker. This is also the same format for commands if you were to use repository locally.

- Generate data for benchmark results presented in paper: `make graphs`.

  This runs all benchmarks and regenerates table files located in `/eff/misc/code-generation-benchmarks/generate-graphs/tables/`. Filename consists of benchmark name (`count`, `interpreter-state`, ...) and benchmark backend (`generated` for generated OCaml code, `native`, `capabilities`, ...).

  Each file has a header with user-readable information and space-separated list of values: input parameter and execution time ratio with respect to native OCaml version.

  Use `nano` to view benchmark results: `nano misc/code-generation-benchmarks/generate-graphs/tables/loop_benchmarks-generated.table`.

- Run benchmarks and output benchmark data in human-readable format: `make benchmark`.

  This runs the same benchmarks as presented in paper but outputs human readable `bechamel` results. The output of the full result is saved in file `misc/code-generation-benchmarks/display-results/benchmark.expected` and can be visualized using `cat misc/code-generation-benchmarks/display-results/benchmark.expected`.

  Specific benchmarks can be excluded and tried on different parameters by modifying file `misc/code-generation-benchmarks/benchmark-suite/benchmark_config.ml`. Multicore OCaml specific config is available in `misc/code-generation-benchmarks/benchmark-suite/config.multicore.ml`.

- Compile custom `eff` source code in file `INPUT_FILENAME` using compiler presented in paper with all optimizations and write resulting `OCaml` program to file `OUTPUT_FILENAME` `/eff.exe --no-stdlib --compile-plain-ocaml INPUT_FILENAME > OUTPUT_FILENAME.ml`.

  Command `./eff.exe --no-stdlib --compile-plain-ocaml misc/code-generation-benchmarks/benchmark-suite/loop.eff > out.ml` compiles all loop benchmarks presented in paper.

  Resulting file has a dependency on `ocamlHeader.ml` located in `ocamlHeader/ocamlHeader.ml` and to use it, you have to compile them together `ocaml ocamlHeader/ocamlHeader.ml out.ml` or just manually copy contents from the header to the generated file and use it as-is.

- Extend and change the compiler.

  Full compiler source code is available and users can make their own modifications and test the compiler. We encourage users to use the standard OCaml compiler for this by running `opam switch 4.12; eval $(opam env)`. Running `make` recompiles the `Eff` compiler and running `make generate_benchmarks` regenerates the `Eff` based benchmarks. This recompiles benchmarks to `OCaml` code and outputs a diff of newly compared version and older version. *The return code is the same as the return code of `diff`, so if any changes are made, this command deliberately exits with an error and old files are replaced with new ones.*

WARNING: Docker file system is non persistent, so any changes made in container will disappear after container exits. We suggest using local repository for compiler modifications.

Claims

Claims supported by the artifact:

- Prototype implementation of optimizing compiler as an extension of `Eff`.

  Artifact provides a straightforward way to compile `Eff` source files with experimental version of compiler and using it as a normal OCaml source (see: Compile custom `Eff` source code).

Claims not/partially supported by the artifact:

- Due to virtualization overhead and lowered benchmark runs, performance claims might not be reproducible in docker environment and the measurement results suffer from additional noise caused by low number of repeats.

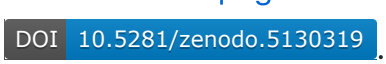
Remarks

This artifact provides an `eff` compiler with support for optimized compilation to `OCaml` source code. Benchmarks consist of `OCaml` code generated by the presented compiler (`ocamlformat` is automatically applied to the resulting code). Command `make generate_benchmarks` invokes the compiler and translates `eff` source code in `misc/code-generation-benchmarks/benchmark-suite/` into `OCaml` source code. Benchmarks presented in the paper are already generated in the artifact.

Running `make test` with the multicore version of OCaml compiler fails due to missing `ocamlformat` library. See section `Extend and change the compiler` on how to switch to standard OCaml.

Artifact links

- Zenodo code release: 

.
- Dockerhub Dockerfile, sha: `886ad60ac677a9879bf434a49380b5b7d9dd7359f93230fffacbbf31acf62444`.
- Artifact information and instructions: page.
- Dockerfile on zenodo: 

.